RAPID PROTOTYPING WITH CRDTS AND REPLIKATIV

CHRISLAIN RAZAFIMAHEFA

SHARED OBJECT

razafima@gmail.com

ABOUT ME

- Founder of ***Shared Object Sarl***
- Past Adventures
  - Java
    - Compiler and VM implementation, J2EE Applications
  - Rails
  - Clojure(Script)

MOTIVATION

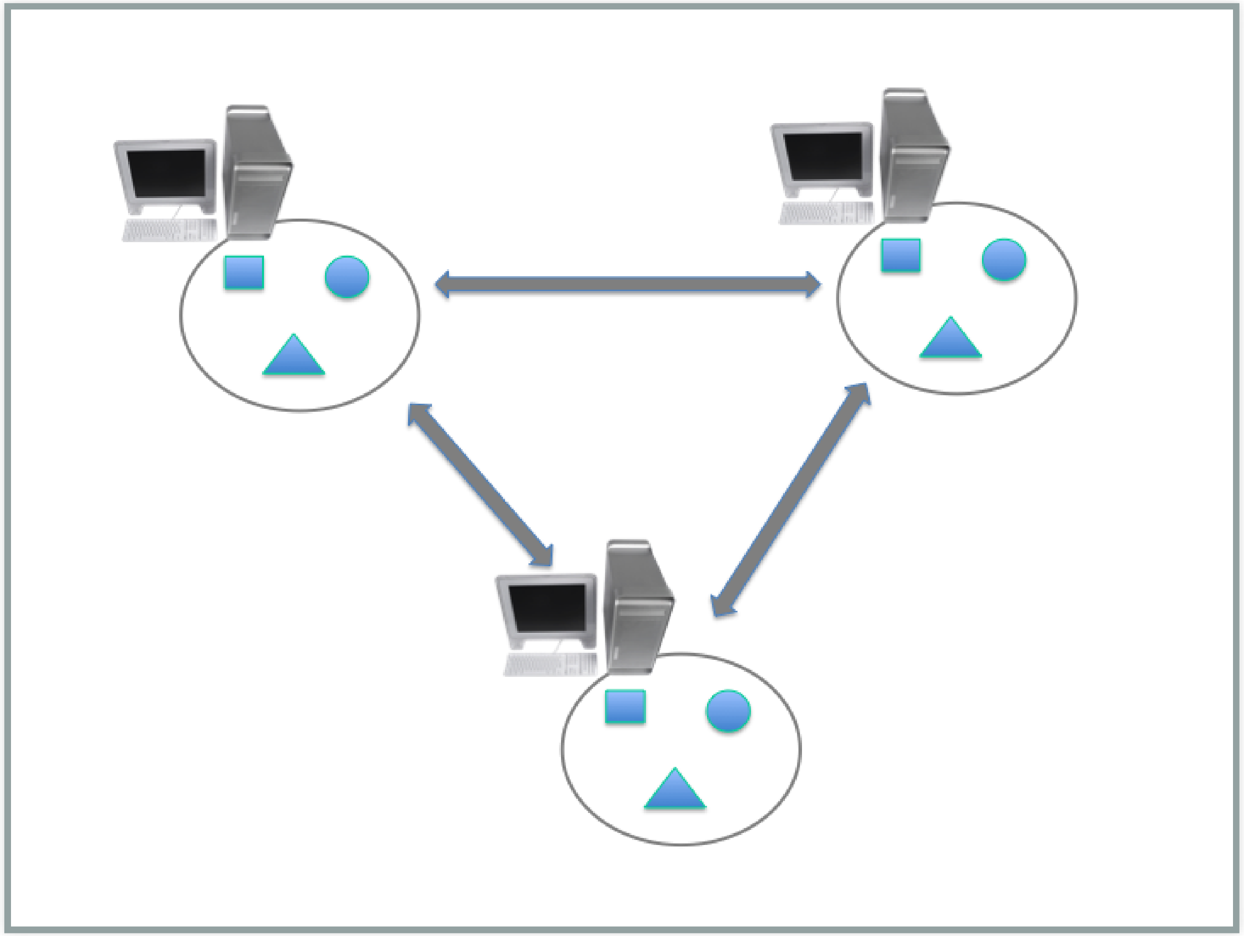
Compared to current approaches...

Is it possible to develop distributed applications...

- which are SIMPLER to build
- and results in better **scalability, performance, robustness**, ...

DISTRIBUTED SYSTEMS

~ Multiple Machines Used For One Purpose



REPLICATION

- Goal

  Have the same data on all replicas

- Why
  - Increase **Availability**
    - Safer systems
  - Reduce **Latency**
    - Faster systems
  - Increase **Throughput**
    - By increasing the number of available nodes
    - Scalabale systems
  - Work **Offline**

CHALLENGES WITH REPLICATION

- How to deal with concurrent UPDATES?
  - Synchronous / Synchronized
    - Strongly consistent
    - Main issues: Slow and Does not scale
  - Asynchronous
    - Fast and Scales
    - Main Issue: CONFLICTS

MAGIC?

- Is there a **magical** solution…
  - which **resolve conflicts** automatically
  - which is also **fast** and **scales**

CRDTS

Conflict-free Replicated Data Type

A comprehensive study of Convergent and Commutative Replicated Data Types *

Marc Shapiro, INRIA & LIP6, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal
Carlos Baquero, Universidade do Minho, Portugal
Marek Zawirski, INRIA & UPMC, Paris, France

Thème COM — Systèmes communicants

Projet Regal

Rapport de recherche n° 7506 — Janvier 2011 — 47 pages

Abstract: Eventual consistency aims to ensure that replicas of some mutable shared object converge without foreground synchronisation. Previous approaches to eventual consistency are ad-hoc and error-prone. We study a principled approach: to base the design of shared data types on some simple formal conditions that are sufficient to guarantee eventual consistency. We call these types Convergent or Commutative Replicated Data Types (CRDTs). This paper formalises asynchronous object replication, either state based or operation based, and provides a sufficient condition appropriate for each case. It describes several useful CRDTs, including container data types supporting both *add* and *remove* operations with clean semantics, and more complex types such as graphs, montonic DAGs, and sequences. It discusses some properties needed to implement non-trivial CRDTs.

TODO INTUITION

- Whatever the order of the operations, whether there are concurrent operations or not, **eventually** all replicas will converge
- Obtained by forcing operations on the data structures to have simple mathematical properties such as commutativity,…

TWO TYPES OF CRDTS

- Operation based

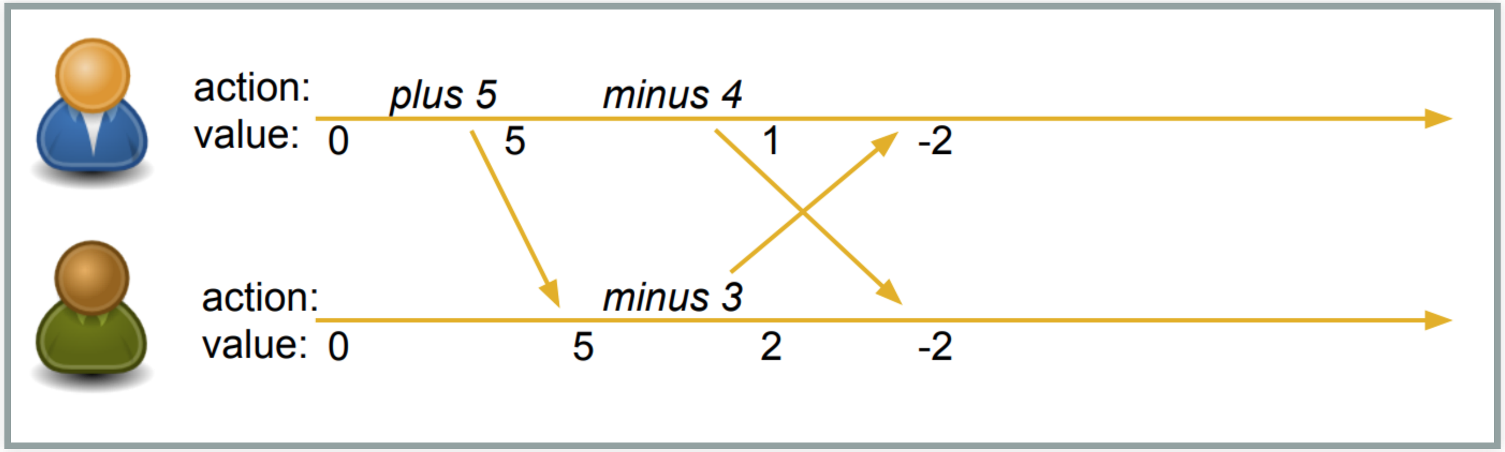
  Operations are exchanged between nodes

- State based

  States are exchanged between nodes

OPERATION BASED

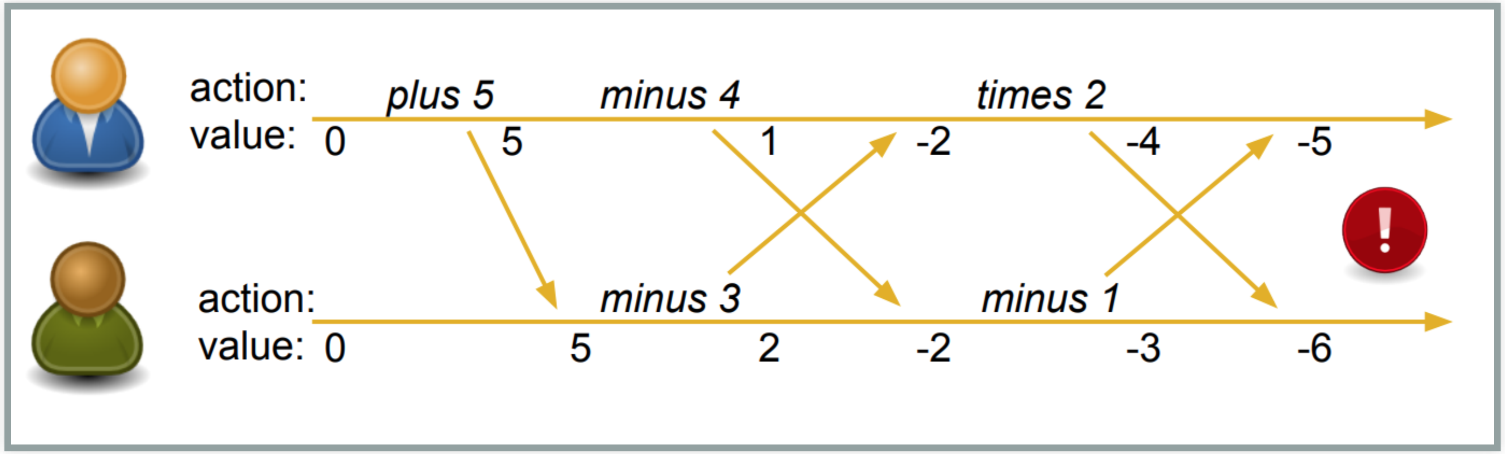
- Ex.: COUNTER



  - (5 - 4 - 3) = (5 - 3 - 4)

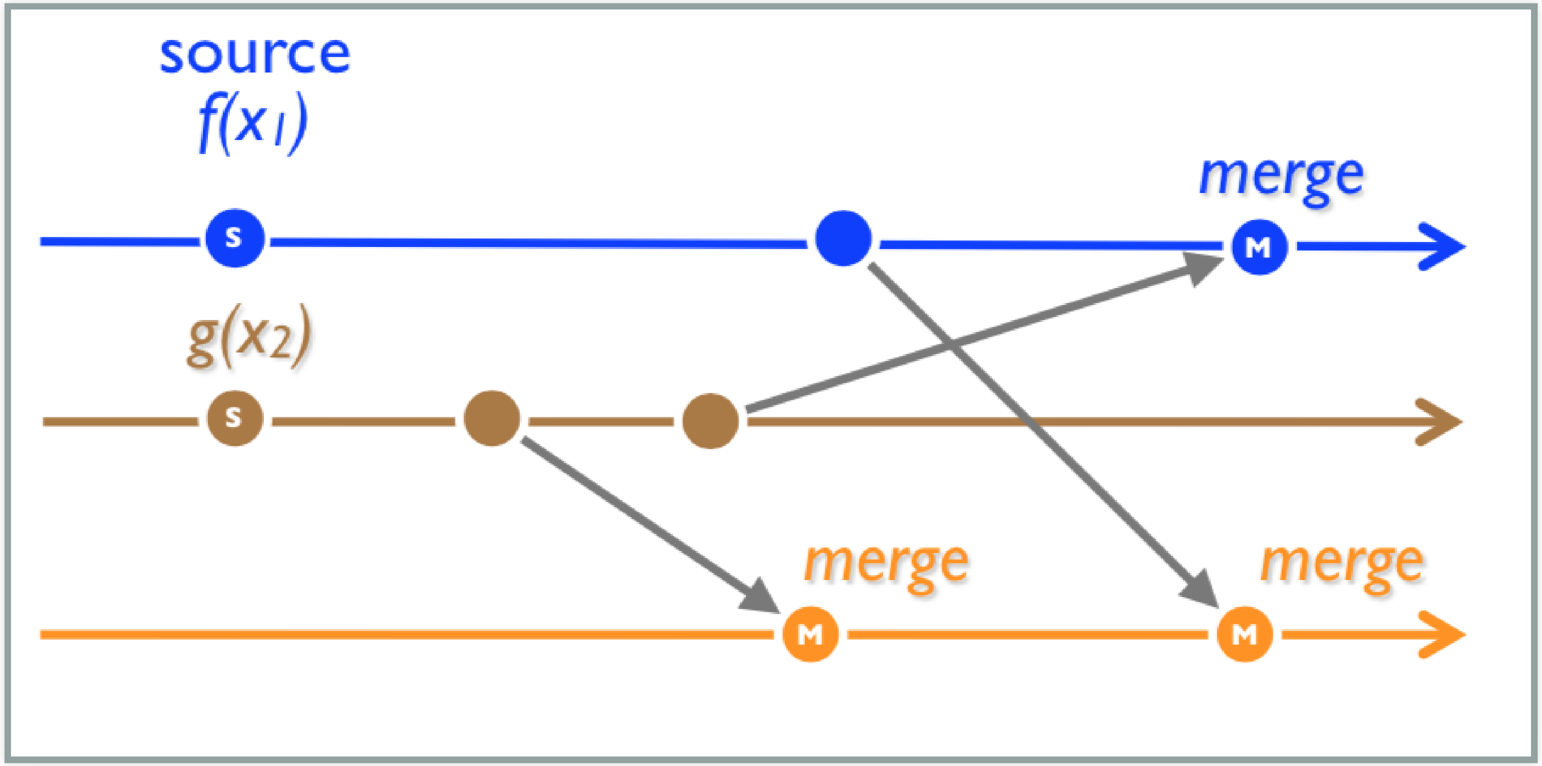
OPERATION BASED

- If we add multiplication...



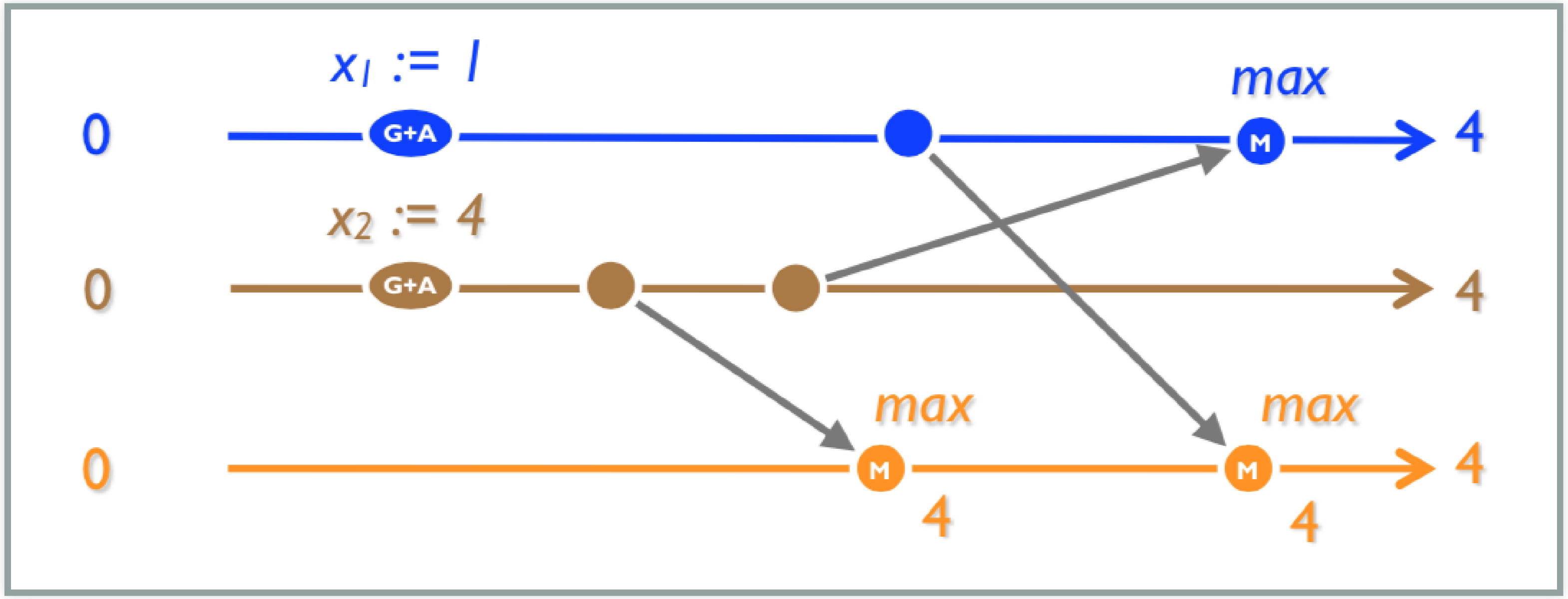
- (5 - 4 - 3) * 2 - 1 != (5 - 3 - 4 - 1) * 2
- **COMMUTATIVITY** is key

STATE BASED



- Locally states are updated by operations *f*, *g*, ...
- States are propagated to remote replicas and **merged**

STATE BASED: MAX EXAMPLE



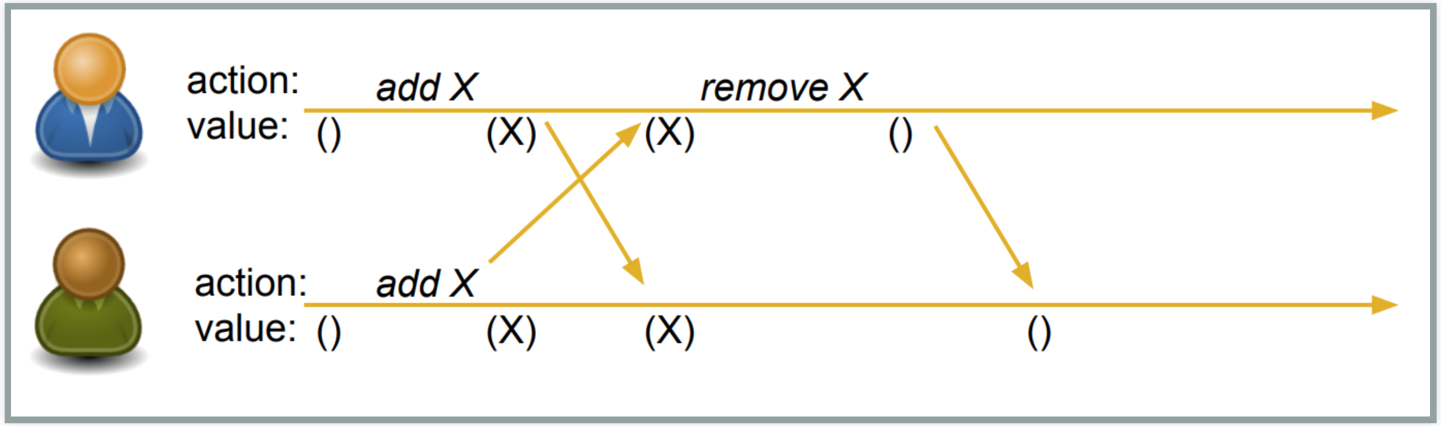
- Local operation: 'set'
- Merge operation: 'max'

STATE BASED: CONVERGENCE

- Only when **merge** is:
  - Commutative: a + b = b + a
  - Associative: (a + b) + c = a + (b + c)
  - Idempotent: a + a + a = a

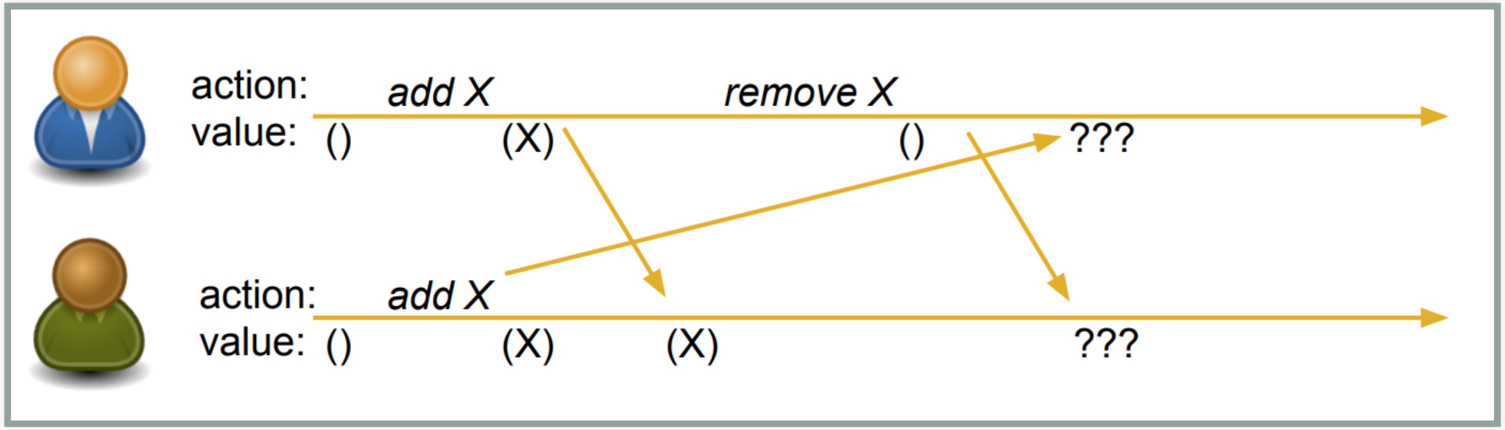
SET

- Naive approach, i.e. like sequential version
  - When lucky:



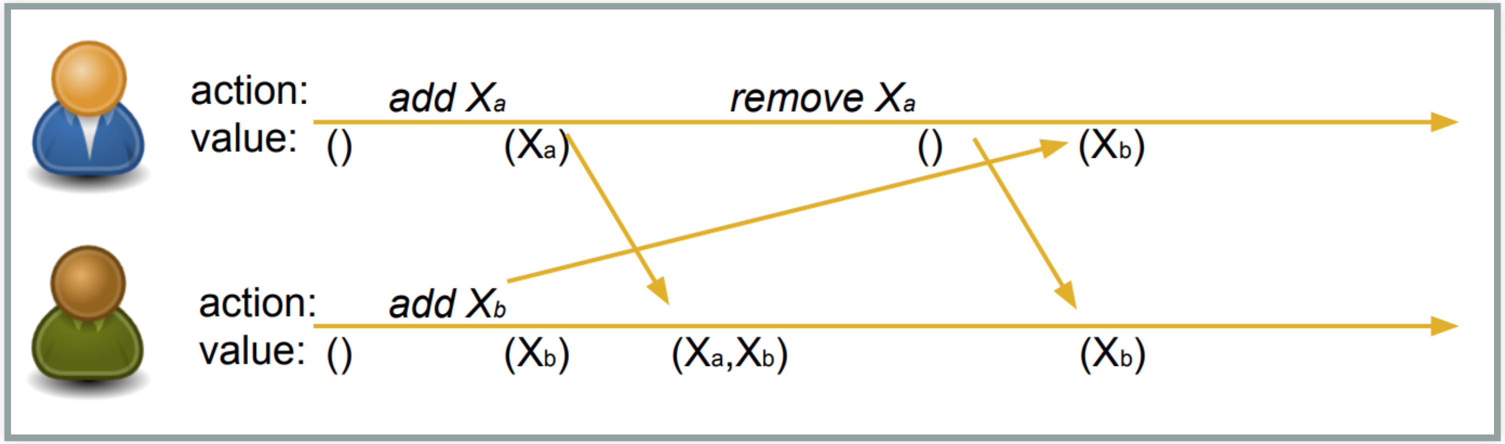
SET

- When Not lucky:



OBSERVED-REMOVE SET

- Add a tag on each replica to uniquely identify set elements



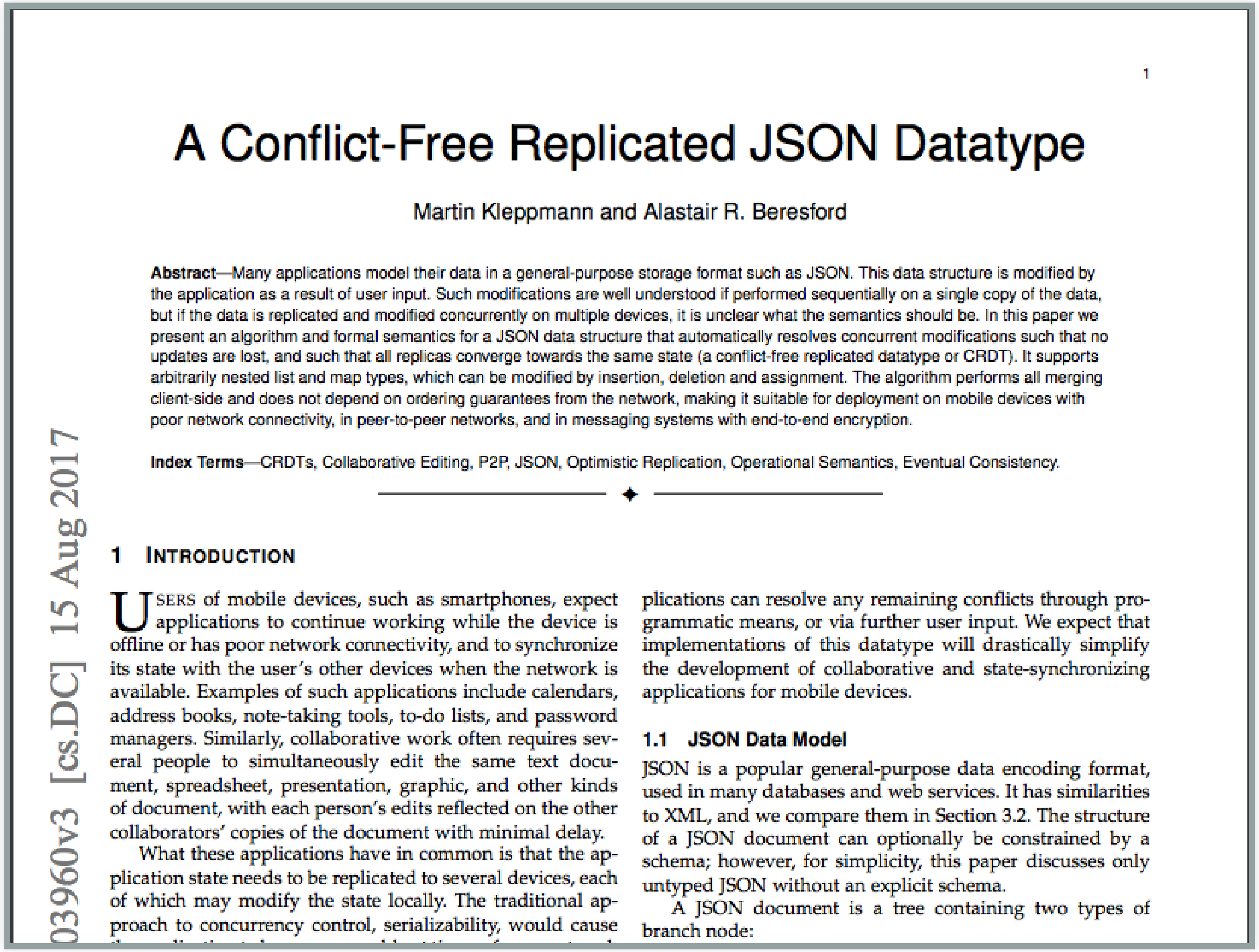
- When concurrent 'add and 'remove, OR-Set favors 'add

AVAILABLE CRDTS

- Counter
- Map
- Set
- Ordered Set
- Graph
- ...

JSON CRDTS

- Based on recent work by Kleppmann & co.



1

A Conflict-Free Replicated JSON Datatype

Martin Kleppmann and Alastair R. Beresford

Abstract—Many applications model their data in a general-purpose storage format such as JSON. This data structure is modified by the application as a result of user input. Such modifications are well understood if performed sequentially on a single copy of the data, but if the data is replicated and modified concurrently on multiple devices, it is unclear what the semantics should be. In this paper we present an algorithm and formal semantics for a JSON data structure that automatically resolves concurrent modifications such that no updates are lost, and such that all replicas converge towards the same state (a conflict-free replicated datatype or CRDT). It supports arbitrarily nested list and map types, which can be modified by insertion, deletion and assignment. The algorithm performs all merging client-side and does not depend on ordering guarantees from the network, making it suitable for deployment on mobile devices with poor network connectivity, in peer-to-peer networks, and in messaging systems with end-to-end encryption.

Index Terms—CRDTs, Collaborative Editing, P2P, JSON, Optimistic Replication, Operational Semantics, Eventual Consistency.

1 Introduction

Users of mobile devices, such as smartphones, expect applications to continue working while the device is offline or has poor network connectivity, and to synchronize its state with the user's other devices when the network is available. Examples of such applications include calendars, address books, note-taking tools, to-do lists, and password managers. Similarly, collaborative work often requires several people to simultaneously edit the same text document, spreadsheet, presentation, graphic, and other kinds of document, with each person's edits reflected on the other collaborators' copies of the document with minimal delay.

What these applications have in common is that the application state needs to be replicated to several devices, each of which may modify the state locally. The traditional approach to concurrency control, serializability, would cause

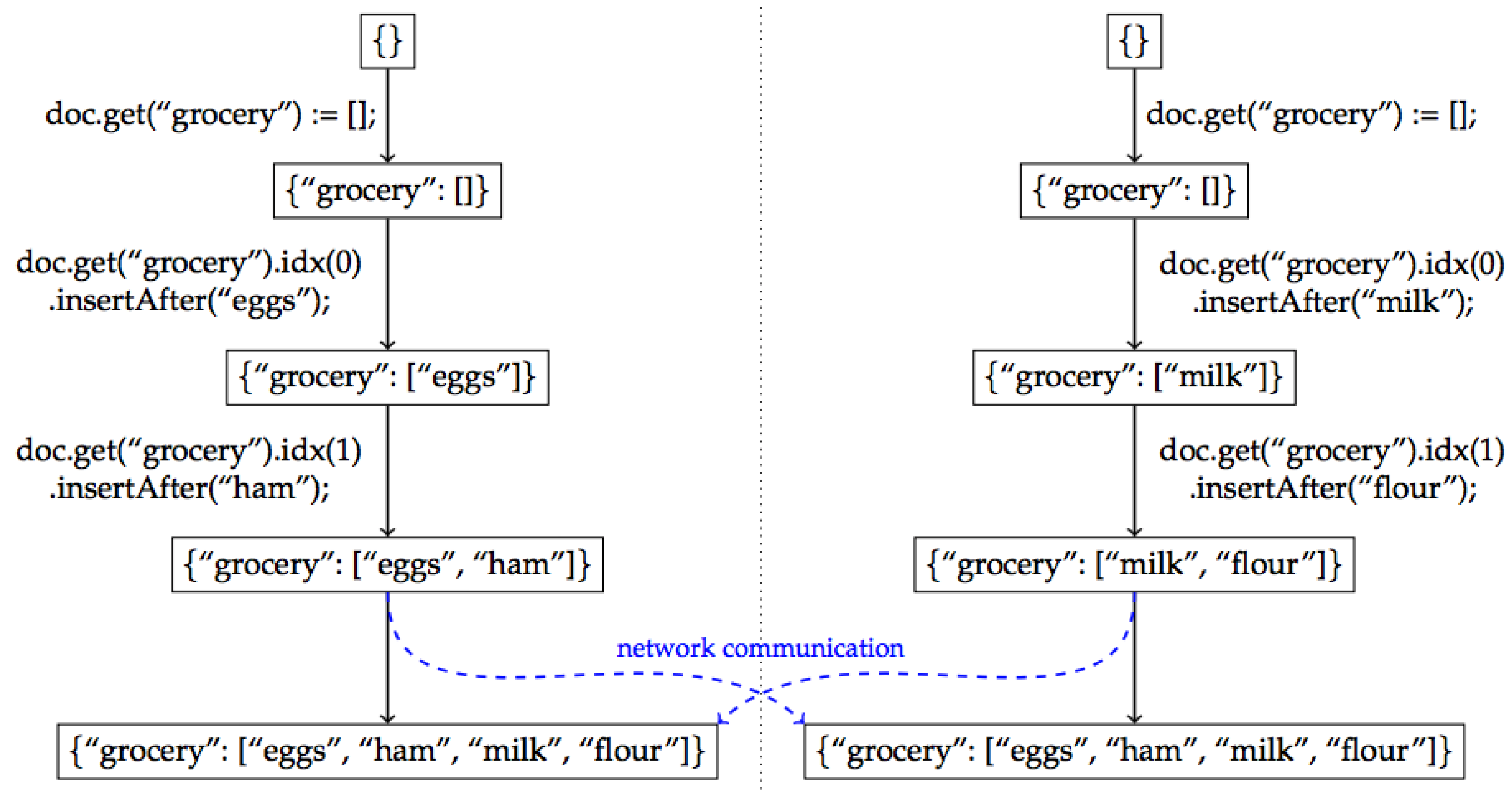
plications can resolve any remaining conflicts through programmatic means, or via further user input. We expect that implementations of this datatype will drastically simplify the development of collaborative and state-synchronizing applications for mobile devices.

1.1 JSON Data Model

JSON is a popular general-purpose data encoding format, used in many databases and web services. It has similarities to XML, and we compare them in Section 3.2. The structure of a JSON document can optionally be constrained by a schema; however, for simplicity, this paper discusses only untyped JSON without an explicit schema.

A JSON document is a tree containing two types of branch node:

EXAMPLE



Two replicas concurrently create ordered lists under the same map key.

- Implementation

  https://github.com/automerge

USAGE IN INDUSTRY

- Riak, Soundcloud, Cassandra
- Programming language: Lasp

REPLIKATIV

- C. Weilbach Motivation
  - Clone app state/data like we clone code with git
  - Free data from cloud and vendor lock-in
  - Ultimate goal: statistical analysis

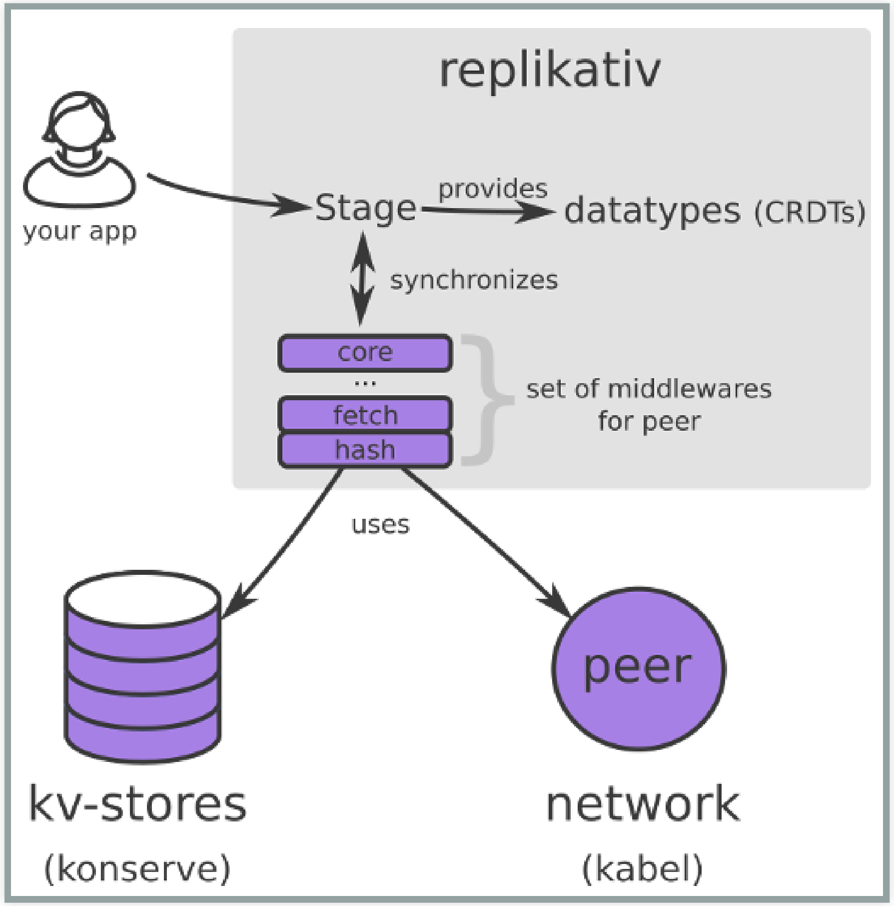
WHAT IS IT?

- For development of distributed applications
- Based on replicated data types: CRDTs
- Can be seen as a distributed database
- Clojure(Script)

NOTICEABLE FEATURES

- Strong eventual consistency
  - No synchronisation
    - No talk to other peers before updating
- Scalability
- Availability -> Work offline
- No distinction between client and server
- Works in browsers, servers and mobile devices
- Works on JS and JVM environments
- Updates propagated automatically in both directions
- Peer-to-peer
- Gossip like protocol
- Functional code-base

ARCHITECTURE



AVAILABLE CRDTS

- map
- set
- cdvcs
- lww
- Soon: EDN i.e JSON in Clojure

USAGE ILLUSTRATION

An app that capture task lengths in a project

- SERVER

```
(defn start-server []
  (let [uri   "ws://127.0.0.1:31778"
        store (<?? S (new-mem-store))
        peer  (<?? S (server-peer S store uri))]
    (run-server #'base-routes {:port 8080})
    (<?? S (start peer))
    (<?? S (chan))))
```

FRONT END

```
(def user "mail:alice@stechuhr.de")
(def ormap-id #uuid "07f6aae2-2b46-4e44-bfd8-058d13977a8a")
(def uri "ws://127.0.0.1:31778")
(defonce val-atom (atom {:captures #{}}))

(defn setup-replikativ []
  (go-try
   S
   (let [store  (<? S (new-mem-store))
         peer   (<? S (client-peer S store))
         stage  (<? S (create-stage! user peer))
         stream (stream-into-identity stage
                                      [user ormap-id]
                                      stream-eval-fns
                                      val-atom)]
     (<? S (s/create-ormap! stage
                            :description "captures"
                            :id ormap-id))
     (connect! stage uri)
     {:store  store
      :stage  stage
      :stream stream
      :peer   peer})))
```

FRONT END

```
(def stream-eval-fns
  {'add (fn [S a new]
          (swap! a update-in [:captures] conj new)
          a)
   'remove (fn [S a new]
             (swap! a update-in [:captures] disj new)
             a)})

(defn add-capture! [state capture]
  (s/assoc! (:stage state)
            [user ormap-id]
            (uuid capture)
            [['add capture]]))
```

UI

```
(defui App
  Object
  (componentWillMount
   [this]
   (om/set-state! this {:input-project ""
                        :input-task ""
                        :input-capture ""}))
  (render [this]
    (let [{:keys [input-project
                  input-task
                  input-capture]} (om/get-state this)
          {:keys [captures]} (om/props this)]
      (html
       [:div
        [:div.widget
         [:h1 "Task Captures"]
         (input-widget this "Project" :input-project)
         (input-widget this "Task" :input-task)
         (input-widget this "Capture" :input-capture)
         [:button
          {:on-click
          (fn [_]
            (let [new-capture {:project input-project
                               :task input-task
                               :capture input-capture}]
              (do
                (add-capture! replikativ-state new-capture)
                (om/update-state! this assoc :input-project "")
                (om/update-state! this assoc :input-task "")
                (om/update-state! this assoc :input-capture ""))))}
          "Add"]]
        [:div.widget                                   ; ⇐
         [:table                                       ; ⇐
          [:tr                                         ; ⇐
           [:th "Project"]                             ; ⇐
           [:th "Task"]                                ; ⇐
           [:th "Capture"]]                            ; ⇐
          (mapv                                        ; ⇐
           (fn [{:keys [project task capture]}]        ; ⇐
             [:tr                                      ; ⇐
              [:td project]                            ; ⇐
              [:td task]                               ; ⇐
              [:td capture]])                          ; ⇐
           captures)]]]))))                            ; ⇐
```

TODO PREPARE VIDEO

CONCLUSION

- Saw overview of CRDTs and REPLIKATIV
- Makes app dev a lot simpler
  - No need to deal with IO anymore
    - No client or server networking dev needed
  - Just work on your local state and the rest is taken care of
  - No big stack to learn
  - No app to install

QUESTIONS?

EXTRA SLIDES

TODO REFERENCES